Sumneet Brar
OS Technical Report
April 29th, 2024

FreeRTOS

FreeRTOS is a “real-time operating system” primarily intended for embedded devices. A real-time operating system is defined as an OS for applications that have critically defined time constraints. This operating system, originally developed by Richard Barry in 2003 and currently managed by Amazon Web Services since 2017, is designed to facilitate programming and deployment on small, low-power edge devices. Its kernel is written in C and it distinguishes itself from traditional operating systems by focusing on real-time performance, efficiency, and simplicity, making it the ideal choice for the constrained environment typical of embedded systems.

Firstly, an embedded device, also known as a single-purpose device, is designed to run dedicated software to perform a single task. An example of such a device is a pacemaker, which continuously monitors and regulates the heart rate of a patient. A pacemaker requires highly specialized software that operates within critical time constraints to ensure it responds accurately and swiftly to the heart's needs. This software is carefully designed to detect minute changes in cardiac activity and adjust pacing in real-time, ensuring that the heart maintains a stable rhythm essential for the patient’s health and safety. Such devices are an effective example of where a specialized OS like FreeRTOS could be used. Along with their time-constrained nature, these devices have limited memory and processing power meaning the software needs to be efficient and capable of running with limited resources. Furthermore, these microdevices come in a wide variety of forms, necessitating an operating system that is both portable and modular to accommodate the diverse hardware configurations. Due to these stringent requirements, FreeRTOS is an ideal choice for many of these devices because of its ability to meet their specific needs.

These systems are different from typical computing systems in the sense that they lack the stringent time constraints, and require operating systems that can handle multiple applications and user demand simultaneously. In contrast, the operating systems in regular consumer computers, such as Windows, macOS, or various distributions of Linux, are designed to support a wide range of applications and user interactions, from simple document editing to complex video rendering and gaming. These systems prioritize user experience, versatility, and an intricate graphical interface, which demand substantial system resources and more sophisticated management of hardware. Additionally, these mainstream operating systems incorporate features for security, user privacy, and connectivity that are adaptable to the evolving technology landscape and user expectations. This contrasts with embedded systems where the focus is primarily on stability, efficiency, and real-time performance, catering to a different set of operational paradigms and constraints. This means that an operating system like FreeRTOS could never replace one of the traditional operating systems but only serve in specialized cases for specific devices.

In terms of process and thread management, FreeRTOS operates with a design geared for simplicity and efficiency, which is essential for embedded systems where resources are limited. Unlike traditional operating systems that support complex multi-threading and multi-processing capabilities, FreeRTOS functions only with tasks, which are analogous to threads in other systems. It itself is designed for only single-core processors or simple multi-core processors where each core runs a separate instance of the OS and thus, is not designed to parallel process like its mainstream counterparts. That being said, FreeRTOS does provide some capabilities that support parallel computing, but not anywhere near the complexity of a traditional operating system as FreeRTOS prioritizes consistency and predictability of programs over performance. With these tasks comes the task management that FreeRTOS handles similarity to other systems. It supports several states in which tasks can reside; these include some traditional ones and a few new ones. Running is the state where a task is actively executing on the CPU; Blocked is a state where a task is waiting for a condition to be met, likely an I/O operation to complete. Suspended is a state where a task is intentionally paused by another task or the OS and will not be scheduled until certain conditions are met and Ready is the state where a task is prepared to run and simply waiting for CPU availability. The last state is when a task has been deleted from the system and can no longer be scheduled. This last state serves to facilitate resource reclamation in a controlled manner. This state management scheme facilitates efficient task transitions, critical for maintaining the system's real-time performance. Compared to more generic operating systems, where process and thread management is designed for a broad range of applications, FreeRTOS is specifically tuned for real-time applications where task prioritization and swift state transitions are the highest priority.

In contrast, regular consumer operating systems like Windows, macOS, or Linux are engineered with a broader scope in functionality and user interaction, supporting an expansive array of processes and threads to accommodate multitasking across varied applications. In essence, the difference in task management strategies between FreeRTOS and general-purpose operating systems illustrates their distinct operational priorities: FreeRTOS emphasizes predictability and efficiency crucial for systems where timing and reliability are paramount, whereas general-purpose operating systems focus on versatility and user experience, necessary for supporting a wide range of desktop and interactive applications. This divergence in design philosophy ensures that each type of system optimally serves its intended environment.

FreeRTOS employs a deterministic and priority-based scheduling algorithm. A deterministic scheduling algorithm is one where the outcomes and operations are entirely predictable, and the timing of task execution can be precisely calculated beforehand. In such algorithms, given a particular set of tasks, resources, and starting conditions, the sequence and execution time of the tasks are always the same. Similarly, Priority-based scheduling is a method where tasks are assigned different priority levels, and the scheduler allocates CPU time to tasks in order of their priority, ensuring that higher-priority tasks receive attention before lower-priority ones.

These two components are fundamental to FreeRTOS's ability to ensure that high-priority tasks receive processing time ahead of lower-priority ones. The scheduler is designed to be pre-emptive by default, meaning that the operating system can interrupt and suspend a currently running task in order to start or resume a higher-priority task, ensuring that critical tasks are executed as soon as they become urgent. However, FreeRTOS can be configured for cooperative multitasking - a scheduling approach where each task voluntarily yields control of the CPU to allow other tasks to run, relying on the tasks themselves to decide when they should give up the CPU, rather than being preemptively interrupted by the operating system. This feature allows high-priority tasks to pre-empt lower-priority tasks, crucial for applications requiring stringent timing accuracy. The primary advantage of this scheduling approach is its predictability, which is essential for real-time tasks. However, it may lead to the starvation of lower-priority tasks if not carefully managed. FreeRTOS addresses potential starvation through the use of time slicing where each task is given a fixed time period before the scheduler automatically switches, ensuring that all tasks receive adequate CPU time.

In contrast, regular operating systems like Windows, macOS, and various Linux distributions employ more complex scheduling algorithms designed to manage a diverse array of user and system tasks simultaneously. These systems typically use a combination of preemptive multitasking and advanced scheduling techniques such as round-robin, multilevel feedback queues, and fair scheduling. In preemptive multitasking, the operating system retains the authority to forcibly interrupt a running process to allocate CPU time to another, higher-priority process, much like the preemptive model in FreeRTOS but often applied across a much broader spectrum of process types and priorities. Advanced algorithms like multilevel feedback queues dynamically adjust the priority of tasks based on their execution characteristics and history, allowing the system to balance responsiveness with fairness across both CPU-bound and I/O-bound processes. For instance, interactive tasks might receive priority boosts to enhance responsiveness, while background tasks are gradually deprioritized to ensure they don't monopolize CPU time, thereby preventing any single process from starving others. These scheduling strategies are geared towards optimizing user experience and system throughput, which are paramount in a general-purpose environment where user interaction and multitasking capabilities take precedence over the strict real-time performance critical in embedded systems managed by FreeRTOS.

In terms of memory management, FreeRTOS provides several options designed to meet the diverse needs of embedded systems. The system allows for both static and dynamic memory allocation. The kernel itself can operate with a fixed amount of memory, reducing the overhead typically associated with dynamic memory management. This is particularly advantageous in embedded environments where memory resources are scarce. The decision to support multiple memory management schemes allows system designers to choose the most appropriate method based on the specific requirements and constraints of their application. Unlike more complex operating systems, FreeRTOS does not implement virtual memory, reflecting its

design for smaller-scale devices with limited memory. Memory protection mechanisms are minimal, focusing instead on efficient and safe memory use within a single-process environment.

FreeRTOS is designed with a pragmatic approach tailored for embedded systems, offering both static and dynamic memory allocation methods to cater to different application needs. Static memory allocation, where memory is allocated at compile time, provides predictability and reliability by eliminating runtime allocation failures—essential in environments where system stability is crucial. This method is particularly beneficial in tightly constrained memory environments, as it ensures efficient use of available resources without the overhead of management during runtime. Conversely, dynamic memory allocation allows for more flexibility by permitting memory to be allocated and freed on-the-fly according to the application's requirements. However, this introduces complexities such as potential fragmentation and the risk of runtime allocation failures, which could be detrimental in critical system operations. For that reason, static memory allocation remains the primary choice. On the other hand, more complex operating systems like Windows and Linux employ advanced memory management techniques that enhance system efficiency and user experience in multi-tasking and multi-user environments. Techniques such partitioning, segmentation, and paging are common as they provide more robust utilization and protection. However, since the primary concern of FreeRTOS is to be compact and used with lower level devices, these complex schemes of memory management tend to be a drawback.

These comprehensive memory management strategies in general-purpose operating systems provide significant advantages in terms of flexibility and efficiency, crucial for supporting the extensive and varied demands of user applications. In contrast, FreeRTOS opts for simpler, more predictable management techniques, focusing on the unique requirements of embedded systems where resource constraints and system predictability are paramount. This contrast in memory management approaches underscores the distinct operational environments and priorities between embedded systems and more versatile computing platforms.

Additionally, the management of I/O devices in FreeRTOS is critical due to its important role in embedded systems. FreeRTOS supports a variety of I/O programming through its portable and adaptable design. It handles I/O devices through direct support in its API, allowing the integration of custom drivers necessary for specific hardware. The simplified approach to I/O enables FreeRTOS to run on a wide array of hardware configurations, making it highly versatile in the field of embedded devices. This versatility is further increased by FreeRTOS’s ability to support both polled and interrupt-driven I/O operations, which provides developers with flexibility in how they design their system’s interactions. Moreover, FreeRTOS facilitates modular I/O handling through its use of the stream buffer and message buffer APIs. These APIs allow for efficient data transfer between tasks and between interrupts, simplifying the development of applications that require complex data handling capabilities, such as networking and buffered communication. The design of

FreeRTOS also considers the constraints and requirements of real-time operations by enabling priority-based task scheduling even within I/O operations. This means that tasks waiting on I/O operations can be prioritized based on their urgency, thus ensuring that critical tasks are not unduly delayed by less critical tasks. This integration of I/O management with the real-time scheduler enhances the overall responsiveness and predictability of the system, which are essential attributes in many embedded applications like automotive controls, industrial automation, and other time-sensitive applications. The management of I/O devices in FreeRTOS is not only robust and flexible but also intricately designed to meet the nuanced demands of varied embedded system applications. Through a combination of direct API support, flexible I/O handling techniques, and integrated real-time capabilities, FreeRTOS provides a comprehensive solution that can adapt to the specific needs of diverse embedded environments, from simple devices to complex industrial systems.

Comparing FreeRTOS to more common operating systems, it distinguishes itself through its stringent focus on real-time capabilities and minimal overhead. This specialization makes it an interesting study in efficiency and adaptability, particularly valuable in the context of embedded systems. As we advance into an increasingly connected world, the principles driving FreeRTOS offer critical insights into how operating systems can be streamlined for maximum efficiency and reliability. This comprehensive examination of FreeRTOS highlights its unique aspects and implementations, revealing how it is designed to meet the demands of its specific application area effectively. By focusing on real-time performance, FreeRTOS not only fulfills the needs of embedded systems but also provides a framework for understanding the broader implications of operating system design for specialized environments.

References

1. FreeRTOS kernel fundamentals - FreeRTOS. (2024). Amazon.com. https://docs.aws.amazon.com/freertos/latest/userguide/dev-guide-freertos-kernel.html
2. FreeRTOS FAQs – Real Time Operating System for Microcontrollers – Amazon Web Services. (2017). Amazon Web Services, Inc. https://aws.amazon.com/freertos/faqs/#:~:text=FreeRTOS%20is%20an%20open%20source,secure%2C%20connect%2C%20and%20manage.
3. FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. (2023, December 15). FreeRTOS. https://www.freertos.org/index.html
4. Wikipedia Contributors. (2024, March 22). FreeRTOS. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/FreeRTOS
5. FreeRTOS Overview - ESP32 - — ESP-IDF Programming Guide v5.2.1 documentation. (2016). Espressif.com. https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/freertos.html